# ECS 271 Assignment 3

Shiven Mian - 917802870

**Part 1:**

1. A replay buffer is used in the DQN to ensure that the data it learns from is independent and identically distributed (iid). For a DQN it typically maintains a record of the state of the environment, action to be taken, reward received and the next state it gets into (i.e it maintains record of random self-play games (training)). Every time, a minibatch is then sampled from this buffer for training and to get the loss - the buffer does a random sampling to ensure that the data is IID. The IID is necessary for the learner for proper training, else the loss it gets can be skewed (the minibatch can have bias in it) and training can be slower.

2. The pseudocode to do this is:

```
    def sample(self, batch_size):

        state, action, reward, next_state, done = zip(*random.sample(self.buffer,
batch_size))
        return np.concatenate(state), action, reward, np.concatenate(next_state), done
```

The np.concat operations had to be done because of the way the ReplayBuffer class was coded and the way state is used in the rest of the code. Otherwise, random.sample is used to randomly sample a batch of size batch_size, and zip is just used to gather the state, action, reward and next_state together. ‘Done’ is used to signal the end of an epoch (i.e episode / game).

3. The pseudocode is:

```
q_value = self.forward(state)
action = q_value.argmax(1).data[0]
```

This is simply a forward propagation (i.e one run of the DQN) for the given state. The action is then the index of the max q value that we get (there are 6 actions so there are 6 q values generated). Note that I also modified the forward() function in the code to use cuda, it sped up as a result.

4. Epsilon is basically the probability of choosing a random action. When we are training the network, we wish to strike a tradeoff between exploring new actions and exploiting existing actions. A random, new action basically tries to explore, while line 48 basically exploits (it simply does a forward prop and gets output). We use this random probability to alternate between exploring and exploiting, and ensuring that we both try and explore new values and exploit already trained, existing values.

It is also worth noting that epsilon decays with time. We earlier want to explore new actions (i.e line 57, epsilon is v high), but once we have trained to a certain extent, the network tries to exploit (i.e line 48, epsilon is less so no random actions).

5. The pseudocode is this: (given we already have target_mode, state, reward, action, next_state)

```
q_target_state = target_model(next_state).max(1)[0]

q_state = model(state).gather(1, action.unsqueeze(1)).squeeze(1)

#target_q = reward + (gamma * q_target_state)

target_q2 = reward + (gamma * q_target_state * (1 - done))

#loss = (q_state - Variable(target_q.data)).pow(2).mean()

loss2 = (q_state - target_q2.view(-1)).pow(2).mean()

return loss2
```

Target_model here is an infrequent copy of the model - this is used for predictions with the bellman equation (target_q2). Q_state is the q_value of the respective action, while target_q2 tries approximating it using the Bellman equation (by taking max of target(next_state)), and the TDL is the expectation of squared loss between these two values.

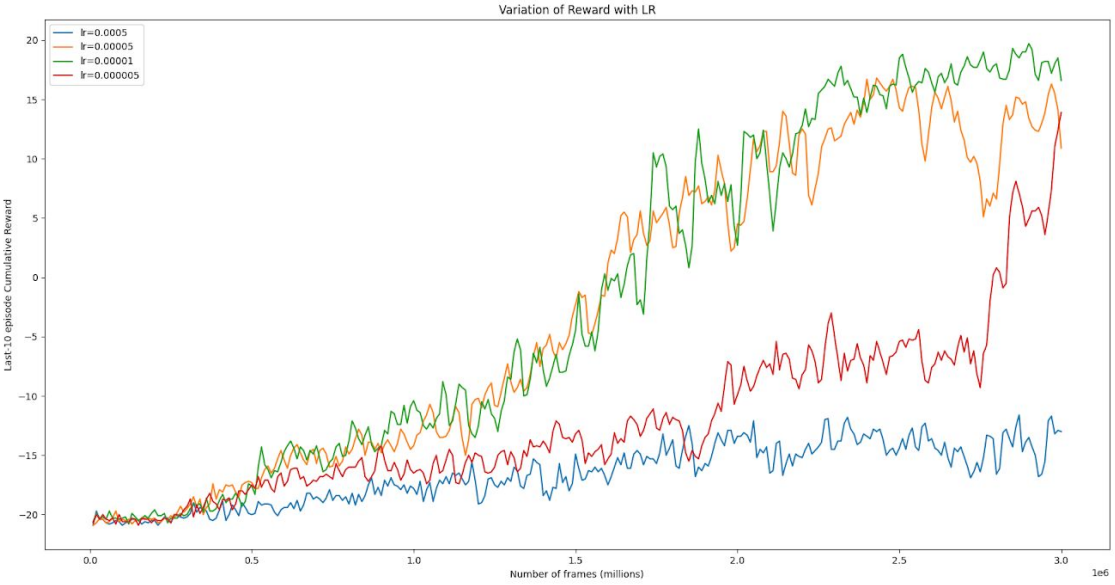
6. My baseline model had the following parameters:

Num_iterations = 3000000 (3 million)
Learning Rate = 0.00001 (1 x 10^-5)
Gamma = 0.99, Other params same

For my best model (lr = 1x10^-5 and gamma = 0.99), my **peak mean cumulative reward** is 20, and as is evident from the jagged plot, it varies over more frames, but it eventually converged within the 16-20 range. **However,** to better test my model, I played a trained model on 10 games with epsilon=0 (i.e no training), and I was getting an average reward of **20.4**.

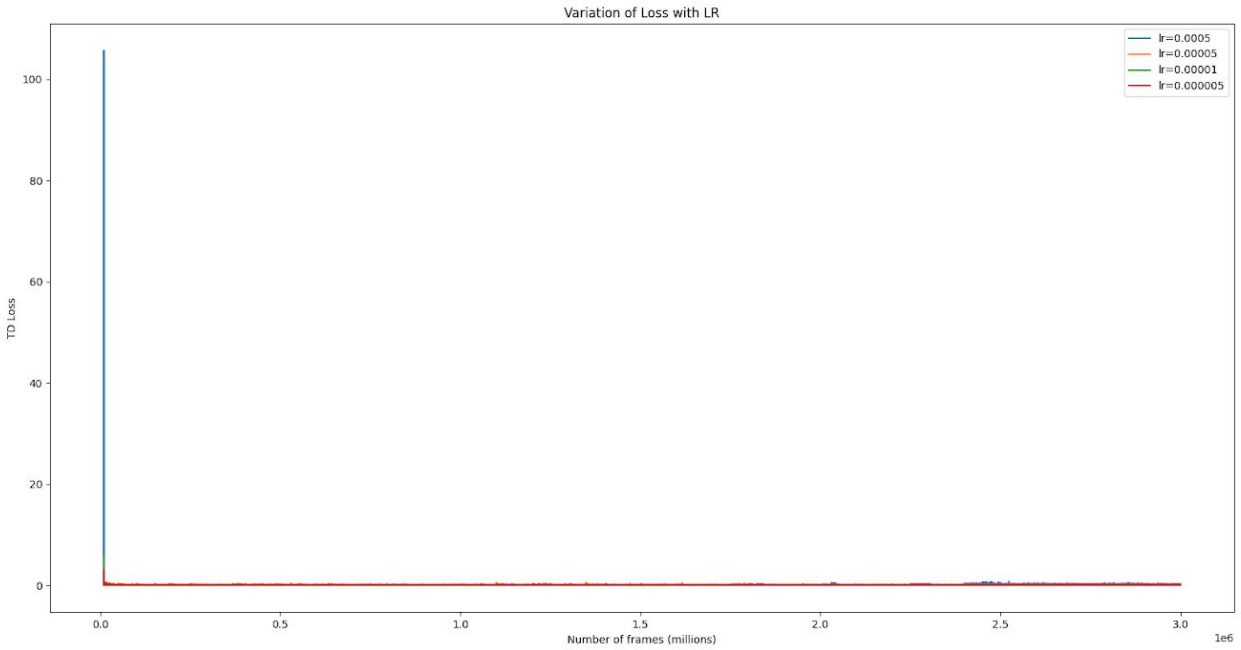
For hyperparam tuning, I tune values of LR and Gamma. These are the plots:

(i) Variation of Reward with LR: I calculate the **mean of the last-10 episode cumulative reward**, per 10000 frames. It is seen that lr=0.00001 performs the best here and grows at a healthy rate. lr=0.00005 has comparable performance but the former's peak mean cumulative reward is higher (20 compared to 16.8). It is seen that when I dramatically increase lr to 0.0005 (50 times more than baseline), the performance is the worst (peak avg cumulative reward is -11.6) because it converges very very quickly at a suboptimal reward and does not optimise much.



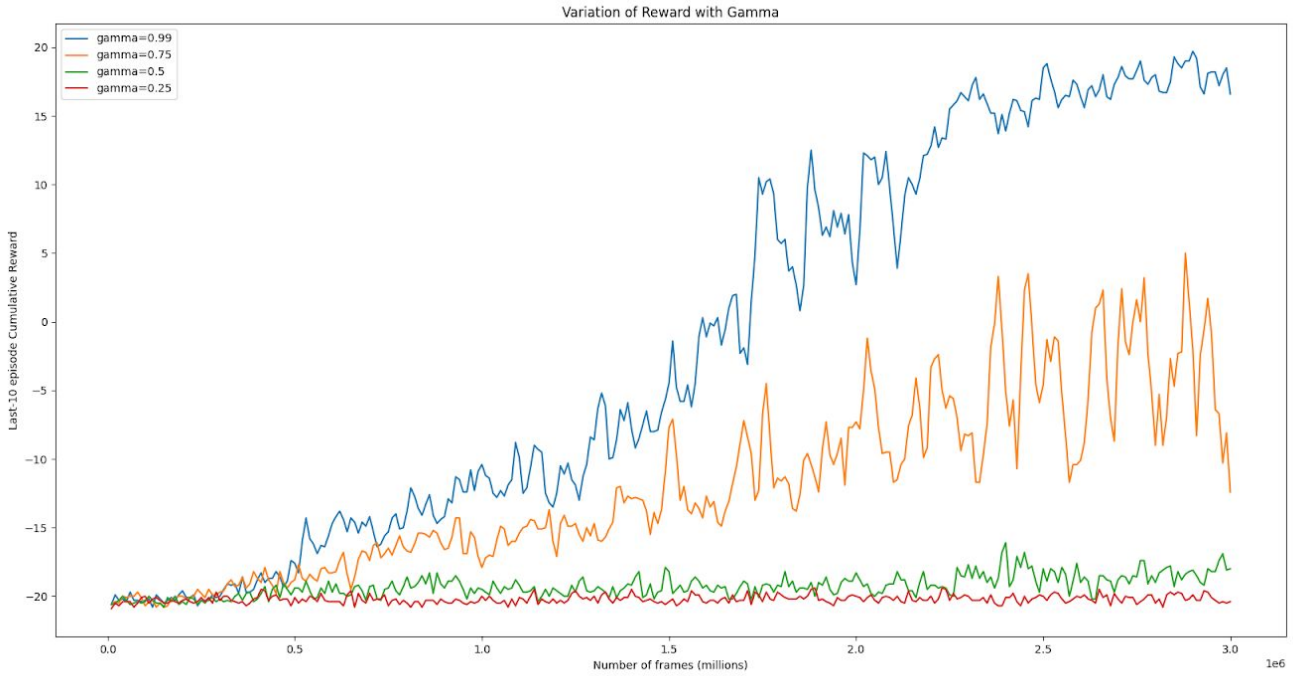

(ii) Variation of Loss with LR:

It is seen that the loss for the smallest lr (0.000005) is the highest, this is possibly because it learns very slowly compared to the others. The others have a comparable loss. Minimum loss of lr = 0.00001 is 4.89 * 10^-5, for 0.00005 it is 4.07 * 10^-5, for 0.000005 it is 9.39*10^-5 and for 0.0005 it is 4.63*10^-5 (plot on next pg)



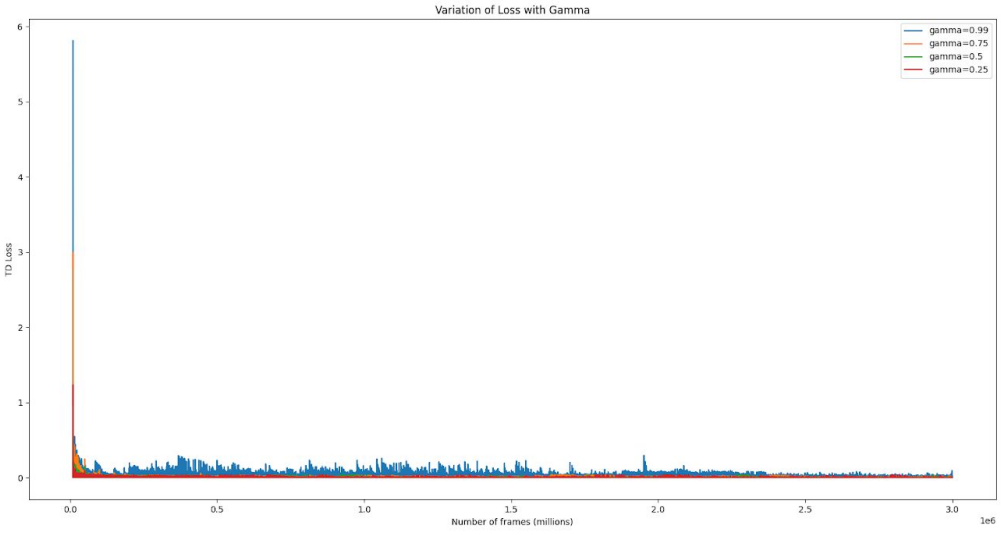

(iii) Variation of reward with gamma

It is clearly seen that as gamma increases, we get better mean cumulative reward, thus my baseline of 0.99 gamma was the best. This is also seen since discounting more means that the target q value becomes lesser, and so loss is more, and so we train better as well. I have talked about loss in the next point.




(iv) Variation of loss with gamma:

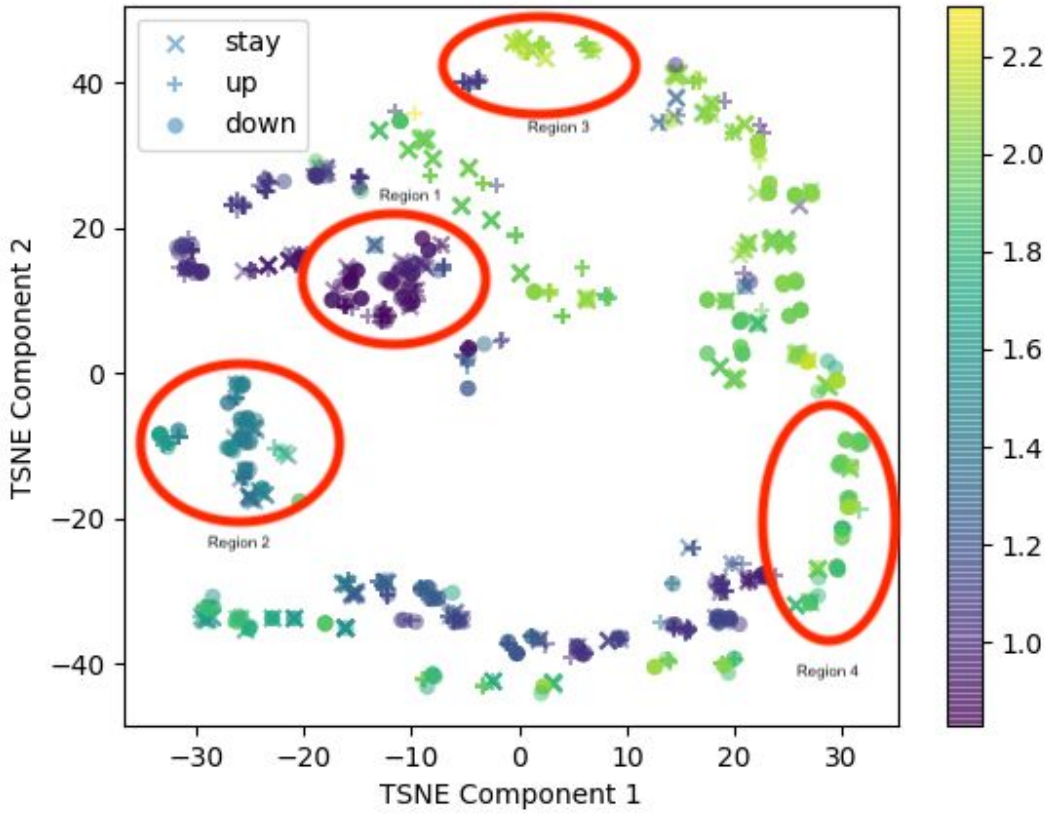
It is seen that the highest gamma also gives the maximum loss over time, as said above, since the bellman equation's q value decreases as gamma increases and so loss is more. This shows that we are also training better since the loss decreases.



**Part 2:**

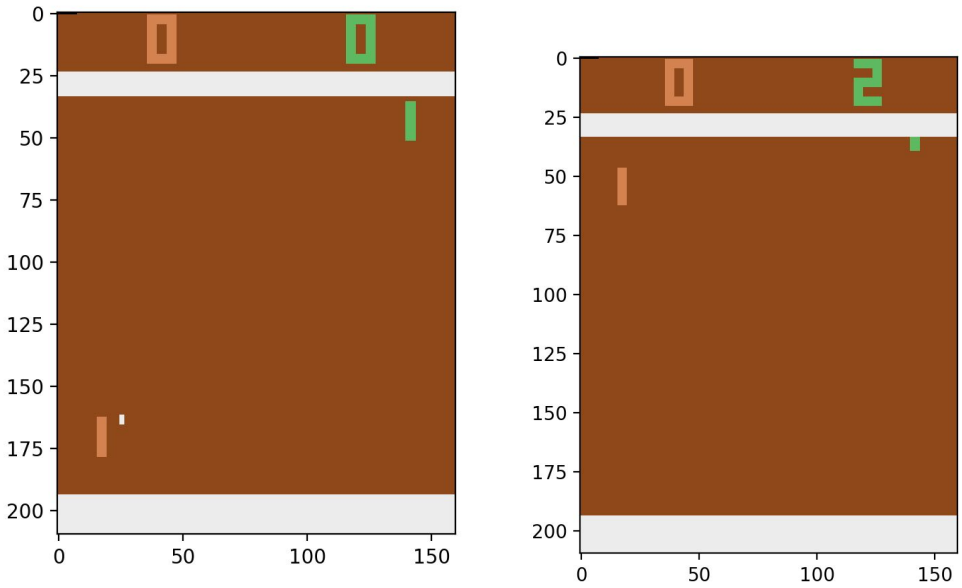
For dimensionality reduction, I tried PCA, TSNE and MDS. I do reduction on the penultimate layer output for 1000 frames (i.e the dimension is (1000,512)). For clustering (i.e coloring) I used the Q values that I get for every frame. I use a separate buffer to store state, action, reward, next_state and also the picture of every frame for analysis.

For me, TSNE with 2 components gave very nice clusters. We can see a few regions here:

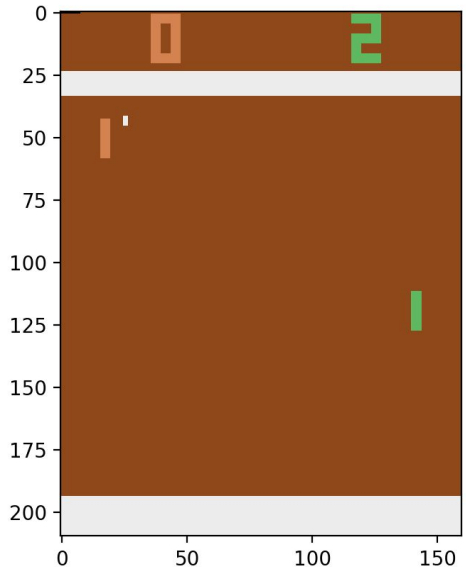


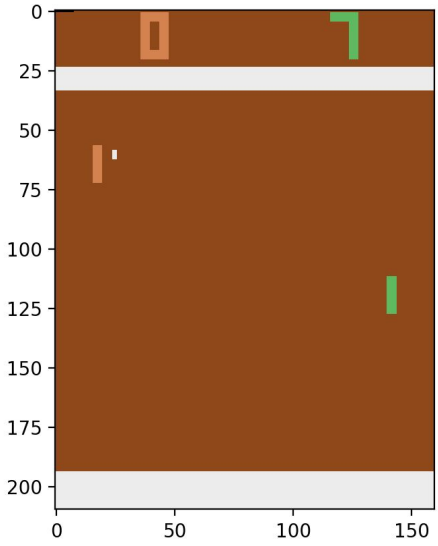

The color bar on the right refers to the Q values using which it is color coded. Here are some interesting properties: (I analyse the positions of the paddles, the actions and the Q value)

1. In Region 1, I see that whenever the ball has gone out of play (i.e a point has ended), and if the opposing paddle is below our paddle, then the choice is always to **go down**. However, in several cases, when the game is in process and the opposing paddle is below our paddle, we choose to **stay.** Here, the left image has Q value of 1.42 and the right image has Q value of 0.96. We can clearly see why in the left image, the paddle chose to stay for now, because the ball is in the direction of the paddle. But in the second image, a new point will be started, so to cover ground, our paddle decides to go down.



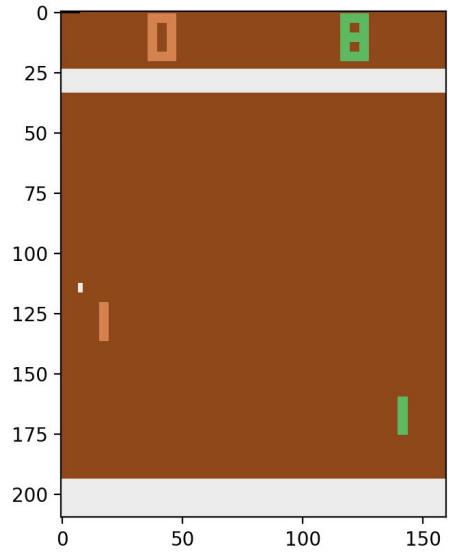

2. In terms of staying, we see a similar pattern in Region 2. In both cases shown below (Q values are 1.54 and 1.42 respectively), the paddle of the opponent is upwards and we choose to **stay** because the ball will rebound and come to our path. For Region 1, in cases of stay and where the ball was present, the opponent paddle was always below ours, so there won't be a rebound. (pics attached next page)




It was also seen that after staying, the action after this was typically to go up, so that our paddle covers the space and the opponent's action.

3. Region 3 seems to be the most descriptive region. In the region, all the instances are of the winning point (pic given). The q-values here were also high: for the below image it is 2.1 and we choose to **stay** because the point is over.




These were the three most descriptive regions I could see (Region 4 did not have entirely clear cut patterns, and even the color coding is not entirely uniform).